SPECpy Documentation

Release 1

Lakhsmipriya Sukumar and Brian Toby

November 01, 2012

CONTENTS

Note that this package requires the Python NumPy and PyEpics packages be installed order to control an instrument. However, if PyEpics is not installed, all routines documented here can still be run. However, in this case EPICS interactions will be simulated and print statements will report what the Python code is attempting to do. Likewise, if PyEpics is installed, but `spec.EnableEPICS()` *is not called (or is called with a value of False), again no communication with EPICS is attempted. This allows scripts to be developed and tested without access to the instrument.*

CHAPTER

ONE

MODULE SPEC: SPEC-LIKE EMULATION

The Python functions listed below are designed to emulate similar commands/macros in SPEC.

1.1 Motor interface routines

Description	Relative	Absolute
move motor	`mvr()`	`mv()`
move motor with wait	`umvr()`	`umv()`
move multple motors		`mmv()`
where is this motor?		`wm()`
where are all motors?		`wa()`
array of motor positions		*A[]*

1.2 Scaler routines

description	command
start and readout scaler after completion	`ct()`
start scaler and return	`count_em()`
wait for scaler to complete	`wait_count()`
read scaler	`get_counts()`

1.3 *Other routines in module spec*

routine	Description
`sleep()`	Delay for a specified amount of time
`EnableEPICS()`	Turns simulation mode on or off
`ShowEnabled()`	Show if use of use of EPICS is available
`DefineMtr()`	Define a motor to be accessed
`DefinePseudoMtr()`	Define pseudo motors from previously defined motors
`GetMtrInfo()`	Retrieves all motor info from a key
`DefineScaler()`	Define a scaler to be accessed
`GetScalerInfo()`	Retrieves all scaler info from an index
`ListMtrs()`	Returns a list of motor symbols
`Sym2MtrVal()`	Retrieves the motor entry key from a symbol
`ExplainMtr()`	Retrieves the motor description from a key or symbol
`ReadMtr()`	Returns the motor position from a key
`PositionMtr()`	Moves a motor
`MoveMultipleMtr()`	Move several motors starting motion together and optionally in increments to keep the motion approximately synchronized.
`GetScalerLastCounts()`	Returns the last set of counts that have been read for a scaler
`GetScalerLastTime()`	Returns the counting time for the last use of a scaler
`GetScalerLabels()`	Returns the labels that have been retrieved for a scaler
`SetMon()`	Set the monitor channel for the scaler
`GetMon()`	Return the monitor channel for the scaler
`SetDet()`	Set the main detector channel for the scaler
`GetDet()`	Return the main detector channel for the scaler
`setCOUNT()`	Sets the default counting time
`initElapsed()`	Initialize the elapsed time counter
`setElapsed()`	Update the elapsed time counter
`setRETRIES()`	Sets the maximum number of EPICS retries
`setDEBUG()`	Sets debugging mode (printing lots of stuff) on or off

1.4 *Global variables*

COUNT defines the default counting time (sec) when ct is called without an argument. Defaults to 1 sec. Use `setCOUNT()` to set this when using `from spec import *`, as setting the variable directly has problems:

This will sort-of work:

```
>>> from spec import *
>>> import spec
>>> spec.COUNT=3
```

however, COUNT in the local namespace will still have the old value.

but this will not work:

```
>>> from spec import *
>>> COUNT=3
```

This fails because the local copy of COUNT gets replaced, but the copy of COUNT actually in the spec module is left unchanged.

S S is a list that contains the last count values measured during the last call to `ct()` or `get_counts()`.

MAX_RETRIES Number of times to retry an EPICS operation (that are nominally expected to work on the first try) before generating an exception. Use `setRETRIES()` to set this or care when changing this (see comment on `COUNT`, in this section.)

DEBUG When set to True lots of print statements to be executed. Use for code development/testing. Use `setDEBUG()` to set this or care when changing this (see comment on `COUNT`, above in this section.)

ELAPSED Contains the time that has elapsed between when the spec module was loaded (or `initElapsed()` was called) and when `setElapsed()` was last called, which happens when motors are moved or counting is done or `sleep()` is called.

1.5 A[]

A As in spec, A[mtr1] provides the current position of mtr1. A is not actually implemented as a global array, but can be indexed as one.

1.6 *Complete Function Descriptions*

The functions available in this module are listed below.

`spec.`**`DefineMtr`**(*symbol*, *prefix*, *comment=''*)

Define a motor for use in this module. Adds a motor to the motor table.

Parameters

- **symbol** (*string*) – a symbolic name for the motor. A global variable is defined in this module's name space with this name, This must be unique; exception specException is raised if a name is reused.
- **prefix** (*string*) – the prefix for the motor PV (ioc:mnnn). Omit the motor record field name (.VAL, etc.).
- **comment** (*string*) – a human-readable text field that describes the motor. Suggestion: include units and define the motion direction.

Returns key of entry created in motor table (str).

If you will use the '' from spec import * '' python command to import these routines into the current module's name space, it is necessary to repeat this command after `DefineMtr()` to import the globals defined within in the top namespace:

Example (recommended for interactive use):

```
>>> from spec import *
>>> EnableEPICS()
>>> DefineMtr('mtrXX1','ioc1:mtr98','Example motor #1')
>>> DefineMtr('mtrXX2','ioc1:mtr99','Example motor #2')
>>> from spec import *
>>> mv(mtrXX1, 0.123)
```

Note that if the second `from ... import *` command is not used, the variables *mtrXX1* and *mtrXX2* cannot be accessed and the final command will fail.

Alternate example (this is a cleaner way to code scripts, since namespaces are not mixed):

```
>>> import spec
>>> spec.EnableEPICS()
>>> spec.DefineMtr('mtrXX1','ioc1:mtr98','Example motor #1')
>>> spec.DefineMtr('mtrXX2','ioc1:mtr99','Example motor #2')
>>> spec.mv(spec.mtrXX1, 0.123)
```

It is also possible to mix the two styles:

```
>>> import spec
>>> spec.EnableEPICS()
>>> spec.DefineMtr('mtrXX1','ioc1:mtr98','Example motor #1')
>>> spec.DefineMtr('mtrXX2','ioc1:mtr99','Example motor #2')
>>> from spec import *
>>> mv(mtrXX1, 0.123)
```

spec.**DefinePseudoMtr**(*inpdict*, *comment=''*)

Define one or more pseudo motors in terms of previously defined motors. Adds the new pseudo motor definition(s) to the motor table.

Parameters

- **inpdict** (*dict*) – defines a dictionary that defines pseudo motor postions in terms of real motor positions and maps pseudo-motor target positions into real motor target positions. Dictionary entries that do not correspond to previously defined motors are used to define new pseudo-motors.
- **comment** (*string*) – a human-readable text field that describes the motor. Suggestion: include units and define the motion direction.

Returns key of entry created in motor table (str).

For computations in the dictionary, motor positions may be referenced in one of two ways, A[mtr] or T[mtr]. A[mtr] provides the actual position of the motor while T[mtr] provides the target position for the move, i.e., the value of the motor or pseudo-motor after the move, if it will be changed. For definitions of pseudo motors, use of A[] is usually correct, but for entries that compute target positions of real motors, one almost always wishes to use T[] to compute from target positions (this is most important for use with `MoveMultipleMtr()`, where multiple target positions are updated prior to any motor movement.). See the examples, below. Note also that these expressions are computed in the spec namespace, so the prefix 'spec.' on motor names (etc.) is not needed.

Note that all the routines in math and numpy are available for use in these calculations (but must be prefixed by math or numpy or np (such as `math.log10()` or `np.exp2()` or `numpy.exp2()` or constant `math.pi`). In addition, for convenience the following functions are also defined without a prefix: `sind()` (sine of angle in degrees), `cosd()` (cosine of angle in degrees), `tand()` (tangent of angle in degrees), `asind()` (inverse sine, returns angle in degrees), `acosd()` (inverse cosine, returns angle in degrees), `atand()` (inverse tangent, returns angle in degrees), `abs()`, `sqrt()` and `exp()`.

Examples:

```
>>> DefineMtr('j1','1idc:j1','sample table N jack')
>>> DefineMtr('j2','1idc:j2','sample table SE jack')
>>> DefineMtr('j3','1idc:j3','sample table SW jack')
>>> spec.DefinePseudoMtr({
...     # define pseudo motor position
...     'jack': '(A[j1] + A[j2] + A[j3])/3.',
...     # map motor movements in terms of pseudo motor target position
...     'j1': 'A[j1] + T[jack] - A[jack]',
```

```
...       'j2': 'A[j2] + T[jack] - A[jack]',
...       'j3': 'A[j3] + T[jack] - ((A[j1] + A[j2] + A[j3])/3)',
...       })
```

The above definition a new pseudo motor, *jack* is defined in terms of three motors that are already defined, *j1*, *j2*, and *j3*. Note that 'T[jack] - A[jack]' (or equivalently 'T[jack] - ((A[j1] + A[j2] + A[j3])/3)', both are used here as a pedagogical example) computes the difference between the target position for jack and its current position and then adds that difference to the positions for *j1*, *j2*, and *j3*, thus, the motors move relative to their initial positions. Note that the comments placed in the input are only a guide to the reader, the fact that 'jack' is new and *j1*, *j2*, and *j3* are defined indicates that *jack* is to be defined.

```
>>>  DefineMtr('samX','1idc:m77','sample X position (mm) + outboard')
>>>  DefineMtr('samZ','1idc:m78','sample Z position (mm) + up')
>>>  DefineMtr('phi','1idc:mphi','sample rotation (deg)')
>>>  spec.DefinePseudoMtr({
...       # define pseudo motor positions
...       'samLX': 'cosd(A[phi])*A[samX] + sind(A[phi])*A[samZ]',
...       'samLZ': '-sind(A[phi])*A[samX] + cosd(A[phi])*A[samZ]',
...       # define motor movements in terms of pseudo motor target position
...       'samX' : 'cosd(T[phi])*T[samLX] - sind(T[phi]) * T[samLZ]',
...       'samZ' : 'sind(T[phi])*T[samLX] + cosd(T[phi]) * T[samLZ]',
...       })
```

In the above definition two new pseudo motors, *samLX* and *samLZ* are defined in terms of three motors that are already defined, *samX*, *samZ*, and *phi*. This maps the axes defined by the sample translations *samX*, *samZ* which are rotated by motor *phi* relative to the diffractometer coordinate system into a static frame of reference. Note that use of T[samLX] and T[samLY] is necessary in the latter expressions, but A[phi] could be used in place of T[phi] as long as one does not try to move phi along with *samLX* and/or *samLY* in a single call to `MoveMultipleMtr()`.

As described for `DefineMtr()`, if you will use the '' from spec import * '' python command to import these routines into the current module's name space, it is necessary to repeat this import command after defining all motors and pseudo motors to import the newly defined global symbols into the top namespace.

`spec.`**`DefineScaler`**(*prefix*, *channels=8*, *index=0*)

Defines a scaler to be used for this module

Parameters

- **prefix** (*string*) – the prefix for the scaler PV (ioc:mnnn). Omit the scaler record field name (.CNT, etc.)
- **channels** (*int*) – the number of channels associated with the scaler. Defaults to 8.
- **index** (*int*) – an index for the scaler, if more than one will be defined. The default (0) is used to define the scaler that will be used when `ct()` is called with one or no arguments.

Example (recommended for interactive use):

```
>>>  from spec import *
>>>  EnableEPICS()
>>>  DefineScaler('id1:scaler1',16)
>>>  DefineScaler('id1:scaler2',index=1)
>>>  ct()
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16]
```

Alternate example (preferred for use in code):

```
>>>  import spec as s
>>>  s.EnableEPICS()
```

```
>>>  s.DefineScaler('ioc1:3820:scaler1',16)
>>>  s.DefineScaler('ioc1:3820:scaler2',index=1)
>>>  s.ct()
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16]
>>>  s.ct(index=1)
[1, 2, 3, 4, 5, 6, 7, 8]
```

spec.**EnableEPICS**(*state=True*)
Call to enable communication with EPICS.

If not called then the module will function in simulation mode only. If the PyEpics module cannot be loaded, then simulation will also be used.

Parameters **state** (*bool*) – if False is specified, then simulation mode is used (default value, True)

spec.**ExplainMtr**(*mtr*)
Show the description for a motor, as defined in `DefineMtr()`

Parameters **mtr** (*various*) – symbolic name for the motor, can take two forms: a motor key or a motor symbol.

Returns motor description (str) or '?' if not defined

spec.**GetDet**(*index=0*)
Return the main detector channel for the scaler or none if not defined. (See `SetDet()`) This is used for ASCAN, etc.

Parameters **index** (*int*) – an index for the scaler, if more than one will be defined (see `DefineScaler()`). The default (0) is used if not specified.

Returns the channel number of the Detector

spec.**GetMon**(*index=0*)
Return the monitor channel for the scaler or none if not defined. (See `SetMon()`) This is used for counting on the Monitor.

Parameters **index** (*int*) – an index for the scaler, if more than one will be defined (see `DefineScaler()`). The default (0) is used if not specified.

Returns the channel number of the Monitor

spec.**GetMtrInfo**(*mtr*)
Return a dictionary with motor information.

Parameters **mtr** (*str*) – a key corresponding to an entry in the motor table. If the value does not correspond to a motor entry, an exception is raised.

Returns dictionary with motor information

spec.**GetScalerInfo**(*index=0*)
returns information about a scaler based on the index

Parameters **index** (*int*) – an index for the scaler, if more than one is be defined (see `DefineScaler()`). The default (0) is used if not specified.

Returns a dictionary with information on the scaler

spec.**GetScalerLabels**(*index=0*)
returns the labels that have been retrieved for a scaler

Parameters **index** (*int*) – an index for the scaler, if more than one is be defined (see `DefineScaler()`). The default (0) is used if not specified.

Returns a list of labels

spec.**GetScalerLastCount**(*index=0*)
returns the last set of counts that have been read for a scaler

Parameters **index** (*int*) – an index for the scaler, if more than one is be defined (see DefineScaler()). The default (0) is used if not specified.

Returns a list of the last counts

spec.**GetScalerLastTime**(*index=0*)
returns the count time for the last read from a scaler

Parameters **index** (*int*) – an index for the scaler, if more than one is be defined (see DefineScaler()). The default (0) is used if not specified.

Returns a single float with the last elapsed time for that scaler (initialized at 0) of the last counts

spec.**ListMtrs**()
Returns a list of the variables defined as motor symbols.

Returns a python list of defined motor symbols (list of str values).

spec.**MoveMultipleMtr**(*mtrposlist*, *nsteps=1*, *wait=True*)
Launch movement of several motors together. If a motor would be moved more than one time (for example because it is referenced in more than on pseudo-motor), only the last move is actually performed. The target for each motor is included in subsequent computations, so that when motor positions are computed from postions of more than one pseudo-motor, the performed move will represent the positions from the cummulative move of all previous motors. To deal with the case where motor speeds or movements are unequal, the requested moves can be broken down into a series of :nsteps: steps, where each motor will be moved an increment of 1/:nsteps: times the total requested change in position. This will not keep the movement on exactly the requested trajectory, but it will stay close.

Parameters

- **mtrposlist** (*list*) – A list of motor keys and target positions, for example [(samLX,1.1),(samLZ,0.25)]
- **nsteps** (*int*) – the number of steps to be used to break down the requested move. The default, 1, means that all motors are launched at the same time for the entire requested movement range, but a value of 2 indicates that all motors will launched to the mid-point of the requested movement range and only after all motors have reached that point, will the subsequent set of moves be started. For :nsteps: > 2,
- **wait** (*bool*) – When *wait* is False, moves are started, but the routine returns immediately, but *wait* is True (default), the routine returns after all motors have stopped moving. If :nsteps: is greater than 1, this parameter is ignored and the routine returns only after all requested moves are completed.

Example:

```
>>> MoveMultipleMtr([(samLX,1.1),(samLZ,0.25)],5,wait=True)
```

spec.**PositionMtr**(*mtr*, *pos*, *wait=True*)
Move a motor

Position a motor associated with mtr to position pos, wait for the move to complete if wait is True, or else return immediately. The function attempts to verify the move command has been acted upon.

Parameters

- **mtr** (*int*) – a value corresponding to an entry in the motor table, as defined in DefineMtr(). If the value does not correspond to a motor entry, an exception is raised.

- **pos** (*float*) – a value to position the motor. If the value is invalid or outside the limits an exception occurs (todo: are hard limits checked?).
- **wait** (*bool*) – a flag that specifies if the move should be completed before the function returns. If False, the function returns immediately.

spec.**ReadMtr**(*mtr*)

Return the motor position associated with the passed motor value.

Parameters **mtr** (*int*) – a key corresponding to an entry in the motor table. If the value does not correspond to a motor entry, an exception is raised.

Returns motor position (float).

spec.**SetDet**(*Detector=None*, *index=0*)

Set the main detector channel for the scaler. The default is to restore this to the initial setting, where this is undefined. This is used for ASCAN, etc.

Parameters

- **Monitor** (*int*) – channel number. If omitted the Monitor is set as undefined. The valid range for this parameter is 0 through one less than the number of channels.
- **index** (*int*) – an index for the scaler, if more than one will be defined (see `DefineScaler()`). The default (0) is used if not specified.

spec.**SetMon**(*Monitor=None*, *index=0*)

Set the monitor channel for the scaler. The default is to restore this to the initial setting, where this is undefined. This is needed for counting on the Monitor.

Parameters

- **Monitor** (*int*) – channel number. If omitted the Monitor is set as undefined. The valid range for this parameter is 0 through one less than the number of channels.
- **index** (*int*) – an index for the scaler, if more than one will be defined (see `DefineScaler()`). The default (0) is used if not specified.

spec.**ShowEnabled**()

Show if use of EPICS is allowed or disabled, see `EnableEPICS()`.

Returns True if PyEpics has been loaded, False otherwise

spec.**Sym2MtrVal**(*mtrsym*)

Converts a motor symbol (as a string) to the motor value (key) as assigned in `DefineMtr()`

Parameters **mtrsym** (*str*) – a motor symbol as supplied in `DefineMtr()`. If the value does not correspond to a motor entry, an exception is raised.

Returns motor value (str).

spec.**count_em**(*count=None*, *index=0*)

Cause scaler to start counting for specified period, but return immediately. On the first use, this will take the scaler out of autocount mode and put it into one-shot mode (this is because if one does not read the scaler shortly after a count when in autocount mode, the scaler returns to autocount and the values are lost.) If put in one-shot mode, then autocount will be restored when the python interpreter is exited.

Counting is on time if count is 0 or positive; Counting is on monitor if count < 0

Parameters

- **count-time** (*float*) – time (sec) to count, if omitted `COUNT` is used (see *Global variables* section)

- **index** (*int*) – an index for the scaler, if more than one will be defined (see `DefineScaler()`). The default (0) is used if not specified.

Returns None

Example:

```
>>> count_em()
>>> # do other commands
>>> wait_count()
>>> get_counts()
```

spec.**ct**(*count=None*, *index=0*, *label=False*)

Cause scaler to count for specified period or to a specified number of counts on a prespecified channel (see `SetMon()`)

Counting is on time if count is 0 or positive; Counting is on monitor if count < 0

Global variable S is set to the count values for the n channels (set in `DefineScaler()`) to provide functionality similar to spec.

Parameters

- **count** (*float*) – time (sec) to count, if omitted `COUNT` is used (see *Global variables* section)
- **index** (*int*) – an index for the scaler, if more than one is defined (see `DefineScaler()`). The default (0) is used if not specified.
- **label** (*bool*) – indicates if counts should be printed along with their labels The default (False) is to not print counts

Returns count values for the channels (see `DefineScaler()`)

Example:

```
>>> ct()
[10000000.0, 505219.0, 359.0, 499.0, 389.0, 356.0, 114.0, 53.0]
>>> SetMon(3)
>>> ct(-1000)
[20085739.0, 1011505.0, 719.0, 1000.0, 781.0, 715.0, 226.0, 105.0]
```

spec.**get_counts**(*wait=False*)

Read scaler with optional delay, must follow count_em

reads count values for the channels (see `DefineScaler()`)

Parameters **wait** (*bool*) – True causes the routine to wait for the scaler to complete; False (default) will read the scaler instananeously

Returns a list of channels values

Example:

```
>>> get_counts()
[1, 2, 3, 4, 5, 6, 7, 8]
```

spec.**initElapsed**()

Initialize the elapsed time counter

spec.**mmv**(*mtrposlist*, *nsteps=1*, *wait=True*)

Launch movement of several motors together. By default, waits for all motion to complete. See the equivalent function, `MoveMultipleMtr()`, for a complete description.

Parameters **mtrposlist** (*list*) – A list of pairs of motor keys and target positions

Example:

```
>>>  mmv([(samLX,1.1),(samLZ,0.25)])
```

spec.**mv**(*mtr*, *pos*)

Move motor without wait

If the move cannot be made, an exception is raised.

Parameters

- **mtr** (*int*) – a value corresponding to an entry in the motor table, as defined in `DefineMtr()`. If the value does not correspond to a motor entry, an exception is raised.
- **pos** (*float*) – a value to position the motor. If the value is invalid or outside the limits, an exception occurs.

Example:

```
>>> mv(samX,0.1)
```

spec.**mvr**(*mtr*, *delta*)

Move motor relative to current position without wait.

If the move cannot be made, an exception is raised.

Parameters

- **mtr** (*int*) – a value corresponding to an entry in the motor table, as defined in `DefineMtr()`. If the value does not correspond to a motor entry, an exception is raised.
- **delta** (*float*) – a value to offset the motor. If the resulting value is invalid or outside the limits, an exception occurs.

Example:

```
>>> mvr(samX,0.1)
```

spec.**setCOUNT**(*count*)

Sets the default counting time, see global variable `COUNT` (see *Global variables* section). Used in `ct()`.

Parameters **count** (*float*) – default time (sec) to count.

spec.**setDEBUG**(*state=True*)

Sets the debug state on or off, see global variable `DEBUG` (see *Global variables* section)

Parameters **state** (*bool*) – DEBUG is initialized as False, but the default effect of *setDEBUG*, if no parameter is specified is to turn the debug state on.

spec.**setElapsed**()

Measure time from the last call to `initElapsed()`. Global variable `ELAPSED` is set to this value. This is called after motors are moved and when counting is done with scalers or `sleep()` is called.

Returns the elapsed time in sec (float)

spec.**setRETRIES**(*count=20*)
Sets the maximum number of times to retry an EPICS operation (that would nominally be expected to work on the first try) before generating an exception. See global variable MAX_RETRIES (in *Global variables* section)

Parameters **count** (*float*) – maximum number of times to retry an EPICS operation. Defaults to 20.

spec.**sleep**(*sec*)
Causes the script to delay for *sec* seconds. This routine gets replaced when plotting is loaded by an alternate routine (see sleepWithYield() in macros._makePlotWin())

Parameters **sec** (*float*) – time to delay in seconds

spec.**umv**(*mtr*, *pos*)
Move motor with wait.

If the move cannot be completed, an exception is raised.

Parameters

- **mtr** (*int*) – a value corresponding to an entry in the motor table, as defined in DefineMtr(). If the value does not correspond to a motor entry, an exception is raised.
- **pos** (*float*) – a value to position the motor. If the value is invalid or outside the limits, an exception occurs.

Example:

```
>>> umv(samX,0.1)
```

spec.**umvr**(*mtr*, *delta*)
Move motor relative to current position with wait.

If the move cannot be completed, an exception is raised.

Parameters

- **mtr** (*int*) – a value corresponding to an entry in the motor table, as defined in DefineMtr(). If the value does not correspond to a motor entry, an exception is raised.
- **delta** (*float*) – a value to offset the motor. If the resulting value is invalid or outside the limits, an exception occurs.

Example:

```
>>> umvr(samX,0.1)
```

spec.**wa**(*label=False*)
Print positions of all motors defined using DefineMtr().

Parameters **label** (*bool*) – a flag that specifies if the list should include the motor descriptions. If omitted or False, the descriptions are not included.

Example:

```
>>> wa()
samX            1.0
samZ            0.0
>>> wa(True)
samX            1.0         sample X position (mm) + outboard
samZ            0.0         sample Z position (mm) + up
```

spec.**wait_count**()
Wait for scaler to finish, must follow count_em

Returns None

Example:

```
>>>  wait_count()
```

spec.**wm**(**mtrs*)
Read out specified motor(s).

Arguments one or more motor table entries that are defined in DefineMtr().

Returns a single float if a single argument is passed to wm. Returns a list of floats if more than one argument is passed.

Example:

```
>>> wm(samX,samZ)
[1.0, 0.0]
```

CHAPTER

TWO

MODULE MACROS: ADDITIONAL SPEC-LIKE EMULATION

Python functions listed below are designed to implement functionality similar to that in spec.

General purpose routines	Description
specdate()	Returns the date/time formated like Spec
SetScanFile()	Open a file for scan output
ascan()	Scan a single motor on a fixed range
dscan()	Scan a single motor on a range relative to current position
RefitLastScan()	Fit a user-supplied function to a user-supplied function
SendTextEmail()	Sends an e-mail message to one or more addresses

2.1 *Logging*

An important set of configuration parameters is that which determine what values are recorded. During data collection, for example, after each ascan() or dscan() data point. Also, for use in defining macros, the values can also be saved to a log file using write_logging_parameters().

Logging routines	Description
init_logging()	Initializes the list of items to be reported
show_logging()	Displays a list of the items that will be logged
make_log_obj_PV()	Define Logging Object that records a PV value
make_log_obj_Global()	Define Logging Object that records a global variable
make_log_obj_PVobj()	Define Logging Object that records a value from a PVobj object
make_log_obj_motor()	Define Logging Object that records a motor position.
make_log_obj_scaler()	Define Logging Object that records a scaler channel value.
log_it()	Adds a Logging Object to the list of items to be reported
add_logging_PV()	Adds a PV to the list of items to be reported
add_logging_Global()	Adds a Global variable to the list of items to be reported
add_logging_PVobj()	Adds a PV object to the list of items to be reported
add_logging_motor()	Adds a motor reference to the list of items to be reported
add_logging_scaler()	Adds a scaler channel to the list of items to be reported
write_logging_header()	Writes a header line with labels for each logged item
write_logging_parameters()	Write the current value of each logged variable

Two examples for setting up logging (new method):

```
>>> import macros
>>> macros.init_logging()
```

```
>>> GE_prefix = 'GE2:cam1:'
>>> macros.log_it(macros.make_log_obj_PV('GE_fname',GE_prefix+"FileName",as_string=True))
>>> macros.log_it(macros.make_log_obj_PV('GE_fnum',GE_prefix+"FileNumber"))
>>> macros.log_it(macros.make_log_obj_motor(spec.samX))
>>> macros.log_it(macros.make_log_obj_scaler(9))
>>> macros.log_it(macros.make_log_obj_Global('var S9','spec.S[9]'))
>>> macros.log_it(macros.make_log_obj_PV('p1Vs',"1idc:m64.RBV"))
```

Note that the *make_log_obj_scaler* and *make_log_obj_Global* calls above will record the same value (though with different headings), but the *make_log_obj_scaler* is a better choice as the second option could produce the wrong value if use of a second scaler is later added to a script.

Old method (does the same as the previous) is:

```
>>> import macros
>>> macros.init_logging()
>>> GE_prefix = 'GE2:cam1:'
>>> macros.add_logging_PV('GE_fname',GE_prefix+"FileName",as_string=True)
>>> macros.add_logging_PV('GE_fnum',GE_prefix+"FileNumber")
>>> macros.add_logging_motor(spec.samX)
>>> macros.add_logging_scaler(9)
>>> macros.add_logging_Global('var S9','spec.S[9]')
>>> macros.add_logging_PV('p1Vs',"1idc:m64.RBV")
```

Example for use of logging in a script:

```
>>> mac.write_logging_header(logname)
>>> spec.umv(spec.mts_y,stY)
>>> for iLoop in range(nLoop):
>>>     spec.umvr(spec.mts_y,dY)
>>>     count_em(Nframe*tframe)
>>>     GE_expose(fname, Nframe, tframe)
>>>     wait_count()
>>>     get_counts()
>>>     mac.write_logging_parameters(logname)
>>> mac.beep_dac()
```

This code step-scans motor mts_y. It writes a header to the log file at the beginning of the operation and then logs parameters after each measurement. Measurements are done in GE_expose and the default scaler, which are run at the same time.

Note that it can be useful to put differing sets of logging configurations into files where they can be invoked as needed using execfile(xxx.py) [where xxx.py is the name of the file to be read]. Do not use import for this task because import will process the file when it is referenced first, but will not do anything if one attempts to import the file again (to reset values back after a different setting has been used). One must use reload to force that.

2.2 *Plotting*

Similar to logging, it is also possible to designate that values can be plotted as part of a script. A Logging Object (from the `make_log_obj_...()` routines) is needed for each item that will be plotted.

Plotting routines	Description
make_log_obj_PV()	Define Logging Object that records a PV value
make_log_obj_Global()	Define Logging Object that records a global variable
make_log_obj_PVobj()	Define Logging Object that records a value from a PVobj object
make_log_obj_motor()	Define Logging Object that records a motor position.
make_log_obj_scaler()	Define Logging Object that records a scaler channel value.
DefineLoggingPlot()	Creates a plot (if needed) or tab on tab to display values and register items to be plotted.
UpdateLoggingPlots()	Read and display all parameters added to plot in DefineLoggingPlot().
InitLoggingPlot()	Clear out plotting definitions from previous calls to DefineLoggingPlot().

Examples:

```
>>> macros.DefineLoggingPlot(
...     'I vs pos',
...     macros.make_log_obj_motor(spec.samX),
...     macros.make_log_obj_scaler(2),
...     )
>>> spec.umv(spec.samX,2)
>>> for iLoop in range(30):
...    spec.umvr(spec.samX,0.05)
...    spec.ct(1)
...    macros.UpdateLoggingPlots()
```

In the above example, a scaler channel is read and plotted against a motor position.

```
>>> macros.DefineLoggingPlot(
...     'I vs time',
...     macros.make_log_obj_Global('time (sec)','spec.ELAPSED'),
...     macros.make_log_obj_scaler(2),
...     macros.make_log_obj_scaler(3),
...     )
>>> spec.initElapsed()
>>> for iLoop in range(30):
...    spec.ct(1)
...    macros.UpdateLoggingPlots()
```

In the above example, two scaler channels are plotted against elapsed time.

2.3 *Monitoring*

Monitoring of PVs is used to record values of selected PVs when any designated PV changes. Optionally, only when that PV changes to a specific value or the recording can be limited to not occur more than a maximum frequency. It may be best to perform monitoring in a process separate from the one making changes to EPICS PVs.

Monitoring routines	Description
DefMonitor()	Set up a PV to be monitored
StartAllMonitors()	Start the monitoring operation

Monitor definition examples:

```
>>> spec.EnableEPICS()
>>> macros.DefMonitor('/tmp/tst','1ide1:m1.VAL',
...                ('1id:scaler1.S2','1id:scaler1.S3','1ide1:m1.RBV','1ide1:m1.VAL')
...                )
>>> macros.StartAllMonitors()
```

This will report the values of four PVs every time that PV 1ide1:m1.VAL is changed.

```
>>> macros.DefMonitor('/tmp/tst','1ide1:m1.RBV',
...                ('1id:scaler1.S3','1ide1:m1.RBV','1ide1:m1.VAL'),
...                pvvalue=0.0)
>>> macros.StartAllMonitors()
```

This will report three PVs, but only when PV 1ide1:m1.RBV is changed to 0.0 (within 0.00001)

```
>>> macros.DefMonitor('/tmp/tst','1ide1:m1.RBV',
...               ('1id:scaler1.S2','1id:scaler1.S3','1ide1:m1.RBV','1ide1:m1.VAL'),
...                delay=1.0)
>>> macros.StartAllMonitors()
```

This will report three PVs, every time that PV 1ide1:m1.RBV is changed, but only a maximum of one change will be reported each second.

2.4 *Macros specific to 1-ID*

These macros reference 1-ID PV's or are customized for 1-ID in some other manner.

1-ID specific routines	Description
`beep_dac()`	Causes a beep to sound
`Cclose()`	Close 1-ID fast shutter in B hutch
`Copen()`	Open 1-ID fast shutter in B hutch
`shutter_sweep()`	Set 1-ID fast shutter to external control
`shutter_manual()`	Set 1-ID fast shutter to manually control
`check_beam_shutterA()`	Open 1-ID Safety shutter to bring beam into 1-ID-A
`check_beam_shutterC()`	Open 1-ID Safety shutter to bring beam into 1-ID-C
`Sopen()`	Same as `check_beam_shutterC()`, bring beam into 1-ID-C
`MakeMtrDefaults()`	Create a file with default motor assignments
`SaveMotorLimits()`	Create a file with soft limits for off-line simulations

2.5 *Complete Function Descriptions*

The functions available in this module are listed below.

`macros.`**`Cclose`**`()`
Close 1-ID fast shutter in B hutch

`macros.`**`Copen`**`()`
Open 1-ID fast shutter in B hutch

`macros.`**`DefMonitor`**(*fileprefix*, *pv*, *monitorlist*, *pvvalue=None*, *delay=None*)
Write values of PVs in monitorlist each time that PV pv changes, values are written to a file named by fileprefix + timestamp optionally, values are written only if the PV is set to value pvvalue (if not None) and optionally only recording the first change in a period of delay seconds (if not None):

Monitoring starts when `StartAllMonitors()` is called.

Parameters

- **fileprefix** (*str*) – defines name of file to use
- **pv** (*str*) – PV to monitor
- **monitorlist** (*list*) – list of PVs to report

- **pvvalue** (*?*) – report monitored PV only if this value is obtained
- **delay** (*float*) – do not log changes more frequently than this frequency in seconds

see *Monitoring* for an example of use.

macros.**DefineLoggingPlot**(*tablbl*, *Xvar*, **args*)

Creates a plot window (if needed) or tab on plot to display values. Parameters include a label for the tab, a Logging Object that will be used as an x-value and as many Logging Object as desired (minimum 1) that will be define y-values. Each time this routine is called, a new plot tab is called. As many plot tabs can be created and populated as desired.

see *Plotting* for an example of use.

Parameters

- **tablbl** (*str*) – a label to place on the plot tab
- **Xvar** (*object*) – a reference to a Logging Object created by `make_log_obj_PV()`, `make_log_obj_Global()`, `make_log_obj_PVobj()`, `make_log_obj_motor()` or `make_log_obj_scaler()`
- **Yvar** (*object*) – a reference to a Logging Object created by `make_log_obj_PV()`, `make_log_obj_Global()`, `make_log_obj_PVobj()`, `make_log_obj_motor()` or `make_log_obj_scaler()`
- **Yvar1** (*object*) – a reference to a Logging Object created by `make_log_obj_PV()`, `make_log_obj_Global()`, `make_log_obj_PVobj()`, `make_log_obj_motor()` or `make_log_obj_scaler()`

class macros.**FitClass**(*x*, *y*)

Defines a prototype class for deriving fitting class implementations. A fitting class should define at least two method: __init__ and Eval.

__init__(x,y) computes a list of very approximate values for the fit parameters, good enough to be used as the starting values in the fit. The number of terms computed determines the number of parameter values that will be fit.

Eval(parms,x) provides the function to be fit.

optionally, Format(parms) is used to return a nicely-formatted text string with the fitted parameters.

Eval(*parm*, *x*)

Evaluate the fitting function and return a "y" value computed for each value in x. Ideally this expression computes all values in a single NumPy expression, but looping is allowed. Both parameters should be lists, tuples or numpy arrays.

Parameters

- **parm** (*list,tuple,etc.*) – parameters in the same order as returned by StartParms()
- **x** (*list,tuple,etc.*) – values of the independent parameter (scanned variable) for evaluation of the function.

Format(*parm*)

This prints the parameters, potentially in a way that explains what they mean. If not overridden, one gets "Parameter values = <list>"

Parameters **parm** (*list,tuple,etc.*) – parameters in the same order as returned by StartParms()

StartParms()

Return the starting parameter values determined in __init__()

class macros.**FitGauss**(*x*, *y*)
Define a function for fitting with a Gaussian.

Parameters are defined as:

index	value
[0]	location of peak
[1]	function value at maximum, less parm[3]
[2]	width as FWHM
[3]	added to all points

Eval(*parm*, *x*)
Evaluate the Gaussian

Format(*parm*)
Prints the parameters

class macros.**FitSawtooth**(*x*, *y*, *Symmetric=True*)
Define a function for fitting with a symmetric or asymmetric saw-tooth function.

Parameters are defined as:

index	value
[0]	location of peak
[1]	function value at maximum
[2]	added to all points
[3]	asymmetric: slope on leading side of peak (+ is rising) symmetric: slope on both sides of peak
[4]	asymmetric: slope on trailing side of peak (+ is falling)

Parameters **Symmetric** (*bool*) – determines if the SawTooth is symmetric (True) or asymmetric (False), meaning that the leading side and the trailing side of the peak can have different slopes.

Eval(*parm*, *x*)
Evaluate the sawtooth function

macros.**InitLoggingPlot**()
Clear out plot definitions from previous calls to DefineLoggingPlot(). Prevents updates from occuring in UpdateLoggingPlot(), but does not delete any tabs or the window.

macros.**MakeMtrDefaults**(*fil=None*, *out=None*)
Routine in Development: Creates an initialization file from a spreadsheet describing the 1-ID beamline motor assignments

Parameters

- **fil** (*str*) – input file to read. By default opens file ../1ID/1ID_stages.csv relative to the location of the current file.
- **out** (*str*) – output file to write. By default writes file ../1ID/mtrsetup.py.new Note that if the default file name is used, the output file must be renamed before use to mtrsetup.py

macros.**RefitLastScan**(*FitClass*, ***kwargs*)
Fit and plot an arbitrary equation to data from the last ascan

Parameters **FitClass** (*class*) – a class that defines a minumum of two methods, one to define a fitting function and the other to determine rough starting values for the fitting function. See FitGauss or FitSawtooth for examples of Fitting classes.

Optional: additional keyword parameters will be passed for the creation of a FitClass object.

Returns an optimized list of parameters or None if the fit fails

Example:

```
>>> macros.RefitLastScan(macros.FitSawtooth)
Parameter values =1.45, 28.5, 1.5, 2.1053
array([  1.44999999,  28.50005241,   1.4999749 ,   2.10525894])
```

or

```
>>> macros.RefitLastScan(macros.FitSawtooth, Symmetric=False)
Parameter values =1.45, 28.5, 1.5, 2.1053, 2.1053
array([  1.44999999,  28.5000524 ,   1.49997491,   2.10525896,   2.10525891])
```

macros.**SaveMotorLimits**(*out=None*)

Routine in Development: Creates an initialization file for simulation use with the limits for every motor PV that is found in the current 1-ID beamline motor assignments. import mtrsetup.py or equivalent first. Scans each PV from 1 to the max number defined.

Parameters **out** (*str*) – output file to write, writes file motorlimits.dat.new in the same directory as this file by default. Note that if the default file name is used, the output file must be renamed before use to motorlimits.dat

macros.**SendTextEmail**(*recipientlist*, *msgtext*, *subject='specpy auto msg'*, *recipientname=None*, *senderemail='1ID@aps.anl.gov'*)

Send a short text string as an e-mail message. Uses the APS outgoing email server (apsmail.aps.anl.gov) to send the message via SMTP.

Parameters

- **recipientlist** (*str*) – A string containing a single e-mail address or a list or tuple (etc.) containing a list of strings with e-mail addresses.
- **msgtext** (*str*) – a string containing the contents of the message to be sent.
- **subject** (*str*) – a subject to be included in the e-mail message; defaults to "specpy auto msg".
- **recipientname** (*str*) – a string to be used for the recipient(s) of the message. If not specified, no "To:" header shows up in the e-mail. This should be an e-mail address or @aps.anl.gov is appended.
- **senderemail** (*str*) – a string with the e-mail address identified as the sender of the e-mail; defaults to "1ID@aps.anl.gov". This should be an e-mail address or @aps.anl.gov is appended.

Examples:

```
>>> msg = 'This is a very short e-mail'
>>> macros.SendTextEmail(['toby@sigmaxi.net','brian.h.toby@gmail.com'],msg, subject='test')
```

or with a single address:

```
>>> msg = """Dear Brian,
...   How about a longer message?
... Thanks, Brian
... """
>>> to = "toby@anl.gov"
>>> macros.SendTextEmail(to,msg,recipientname='spamee@anl.gov',senderemail='spammer@anl.gov'
```

A good way to use this routine is in a try/except block:

```
>>> userlist = ['user@univ.edu','contact@anl.gov']
>>> try:
>>>     macros.write_logging_header(logname)
>>>     spec.umv(spec.mts_y,stY)
>>>     for iLoop in range(nLoop):
>>>         spec.umv(spec.mts_x2,stX)
>>>         for xLoop in range(nX):
>>>             GE_expose(fname, Nframe, tframe)
>>>             macros.write_logging_parameters(logname)
>>>             spec.umvr(spec.mts_x2,dX)
>>>         spec.umvr(spec.mts_y,dY)
>>>     macros.beep_dac()
>>> except Exception:
>>>     import traceback
>>>     msg = "An error occurred at " + macros.specdate()
>>>     msg += " in file " + __file__ + "\n\n"
>>>     msg += str(traceback.format_exc())
>>>     macros.SendTextEmail(userlist, msg, 'Beamline Abort')
```

macros.**SetScanFile**(*outfile=None*)

Set a file for output from ascan, etc. The output is intended to closely mimic what spec produces in ascan and dscan.

Parameters **outfile** (*str*) – the file name to be opened. If not specified, output is sent to the terminal. If the file is new (or is the not specified) a header listing all motors, etc. is printed

macros.**ShowPlots**()

Pause to show plot screens. Call this at the end of a script, if needed.

macros.**Sopen**()

If not already open, open 1-ID-C Safety shutter to bring beam into 1-ID-C. Keep trying in an infinite loop until the shutter opens.

macros.**StartAllMonitors**(*sleep=True*)

Start the monitoring defined in DefMonitor. Optionally delay until control-C is pressed. The control-C operation closes all files and clears the monitoring information.

Parameters **sleep** (*bool*) – if True (default) start an infinite loop of one second delays

see *Monitoring* for an example of use.

macros.**UpdateLoggingPlots**()

Read all current values in plot and display in plots

see *Plotting* for an example of use.

macros.**add_logging_Global**(*txt*, *var*)

Define a global variable to be recorded when `write_logging_parameters()` is called.

Parameters

- **txt** (*str*) – defines a text string, preferably short, to be used when `write_logging_header()` is called as a header for the item to be logged.
- **var** (*str*) – defines a Python variable that will be logged each time `write_logging_parameters()` is called. Note that this is read inside the macros module so the variable must be defined inside that module or must be prefixed by a reference to a module referenced in that module, e.g. spec.S[0]

see *Logging* for an example of use.

macros.**add_logging_PV**(*txt*, *PV*, *as_string=False*)
Define a PV to be recorded when write_logging_parameters() is called.

Parameters

- **txt** (*str*) – defines a text string, preferably short, to be used when write_logging_header() is called as a header for the item to be logged.
- **PV** (*str*) – defines an EPICS Process Variable that will be read and logged each time write_logging_parameters() is called.
- **as_string** (*bool*) – if True, the PV will be translated to a string. When False (default) the native data type will be used. Use of True is of greatest for waveform records that are used to store character strings as a series of integers.

see *Logging* for an example of use.

macros.**add_logging_PVobj**(*txt*, *PVobj*, *as_string=False*)
Define a PVobj to be recorded when write_logging_parameters() is called.

Parameters

- **txt** (*str*) – defines a text string, preferably short, to be used when write_logging_header() is called as a header for the item to be logged.
- **PV** (*epics.PV*) – defines a PyEpics PV object that is connected to an EPICS Process Variable. The PV method .get() will be used to read that PV to log it each time write_logging_parameters() is called.
- **as_string** (*bool*) – if True, the PV value will be translated to a string. When False (default) the native data type will be used. Use of True is of greatest for waveform records that are used to store character strings as a series of integers.

see *Logging* for an example of use.

macros.**add_logging_motor**(*mtr*)
Define a motor object to be recorded when write_logging_parameters() is called. Note that the heading text string is defined as the motor's symbol (see spec.DefineMtr()).

Parameters **mtr** (*str*) – a reference to a motor object, returned by spec.DefineMtr() or defined in the motor symbol. The position of the motor will be read and logged each time write_logging_parameters() is called.

see *Logging* for an example of use.

macros.**add_logging_scaler**(*channel*, *index=0*)
Define a scaler channel to be recorded when write_logging_parameters() is called. Note that the heading text string is defined as the scaler's label (which is read from the scaler when spec.DefineScaler() is run).

Parameters

- **channel** (*str*) – a channel number for a scaler, which can be any value between 0 and one less than the number of channels. The last-read value of that scaler logged each time write_logging_parameters() is called.
- **index** (*int*) – an index for the scaler, if more than one is be defined (see DefineScaler()). The default (0) is used if not specified.

see *Logging* for an example of use.

macros.**ascan**(*mtr*, *start*, *finish*, *npts*, *count*, *index=0*, *settle=0.0*, *_func='ascan'*)
Scan one motor and record parameters set with logging to the scanfile (see func:*SetScanFile*).

Parameters

- **mtr** (*str*) – a reference to a motor object, returned by `spec.DefineMtr()` or defined in the motor symbol.
- **start** (*float*) – starting position for scan
- **finish** (*float*) – ending position for scan
- **npts** (*int*) – number of points for scan
- **count** (*float*) – count time. Counting is on time (sec) if count is 0 or positive; Counting is on monitor if count < 0
- **index** (*int*) – an index for the scaler, if more than one will be defined (see `DefineScaler()`). The default (0) is used if not specified.
- **settle** (*float*) – a time to wait (sec) after the motor has been moved before counting is starting. Default is 0.0 which means no delay

Example:

```
>>> spec.SetDet(2)
>>> macros.ascan(spec.samX,1,2,21,1,settle=.1)
```

It is recommended that if ascan will be run in command line, where python commands are typed into a console window, that ipython be used in pylab mode (`ipython --pylab`).

`macros.`**`beep_dac`**(*beeptime=1.0*)

Set the 1-ID beeper on for a fixed period, which defaults to 1 second uses PV object beeper (defined as 1id:DAC1_8.VAL) makes sure that the beeper is actually turned on and off throws exception if beeper fails

Parameters **beeptime** (*float*) – time to sound the beeper (sec), defaults to 1.0

`macros.`**`check_beam_shutterA`**()

If not already open, open 1-ID-A Safety shutter to bring beam into 1-ID-A. Keep trying in an infinite loop until the shutter opens.

`macros.`**`check_beam_shutterC`**()

If not already open, open 1-ID-C Safety shutter to bring beam into 1-ID-C. Keep trying in an infinite loop until the shutter opens.

`macros.`**`dscan`**(*mtr*, *start*, *finish*, *npts*, *count*, *index=0*, *settle=0.0*)

Relative scan of motor, see func:*ascan*,

Parameters

- **mtr** (*str*) – a reference to a motor object, returned by `spec.DefineMtr()` or defined in the motor symbol.
- **start** (*float*) – starting position for scan, relative to current motor position
- **finish** (*float*) – ending position for scan, relative to current motor position
- **npts** (*int*) – number of points for scan
- **count** (*float*) – count time. Counting is on time (sec) if count is 0 or positive; Counting is on monitor if count < 0
- **index** (*int*) – an index for the scaler, if more than one will be defined (see `DefineScaler()`). The default (0) is used if not specified.
- **settle** (*float*) – a time to wait (sec) after the motor has been moved before counting is starting. Default is 0.0 which means no delay

Example:

```
>>> spec.SetDet(2)
>>> macros.dscan(spec.samX,-1,1,21,1,settle=.1)
```

It is recommended that if dscan will be run in command line, where python commands are typed into a console window, that ipython be used in pylab mode (`ipython --pylab`).

`macros.`**`init_logging`**`()`
Initialize the list of data items to be logged

see *Logging* for an example of use.

`macros.`**`log_it`**`(`*LogObj*`)`
Add a Logging Object into list to be recorded when `write_logging_parameters()` is called.

Parameters **LogObj** (*object*) – a reference to a Logging Object created by `make_log_obj_PV()`, `make_log_obj_Global()`, `make_log_obj_PVobj()`, `make_log_obj_motor()` or `make_log_obj_scaler()`

`macros.`**`make_log_obj_Global`**`(`*txt*, *var*`)`
Define Logging Object that records a global variable

Parameters

- **txt** (*str*) – defines a text string, preferably short, to be used when `write_logging_header()` is called as a header for the item to be logged.
- **var** (*str*) – defines a Python variable that will be logged each time `write_logging_parameters()` is called. Note that this is read inside the macros module so the variable must be defined inside that module or must be prefixed by a reference to a module referenced in that module, e.g. spec.S[0]

see *Logging* for an example of use.

`macros.`**`make_log_obj_PV`**`(`*txt*, *PV*, *as_string=False*`)`
Define Logging Object that records a PV value

Parameters

- **txt** (*str*) – defines a text string, preferably short, to be used when `write_logging_header()` is called as a header for the item to be logged.
- **PV** (*str*) – defines an EPICS Process Variable that will be read and logged each time `write_logging_parameters()` is called.
- **as_string** (*bool*) – if True, the PV will be translated to a string. When False (default) the native data type will be used. Use of True is of greatest for waveform records that are used to store character strings as a series of integers.

see *Logging* for an example of use.

`macros.`**`make_log_obj_PVobj`**`(`*txt*, *PVobj*, *as_string=False*`)`
Define Logging Object that records a value from a PVobj object

Parameters

- **txt** (*str*) – defines a text string, preferably short, to be used when `write_logging_header()` is called as a header for the item to be logged.
- **PV** (*epics.PV*) – defines a PyEpics PV object that is connected to an EPICS Process Variable. The PV method .get() will be used to read that PV to log it each time `write_logging_parameters()` is called.

- **as_string** (*bool*) – if True, the PV value will be translated to a string. When False (default) the native data type will be used. Use of True is of greatest for waveform records that are used to store character strings as a series of integers.

see *Logging* for an example of use.

macros.**make_log_obj_motor**(*mtr*)
Define Logging Object that records a motor position. Note that the heading text string is defined as the motor's symbol (see spec.DefineMtr()).

Parameters **mtr** (*str*) – a reference to a motor object, returned by spec.DefineMtr() or defined in the motor symbol. The position of the motor will be read and logged each time write_logging_parameters() is called.

see *Logging* for an example of use.

macros.**make_log_obj_scaler**(*channel*, *index=0*)
Define Logging Object that records a scaler channel value. Note that the heading text string is defined as the scaler's label (which is read from the scaler when spec.DefineScaler() is run).

Parameters

- **channel** (*str*) – a channel number for a scaler, which can be any value between 0 and one less than the number of channels. The last-read value of that scaler logged each time write_logging_parameters() is called.
- **index** (*int*) – an index for the scaler, if more than one is be defined (see DefineScaler()). The default (0) is used if not specified.

see *Logging* for an example of use.

macros.**show_logging**()
Show the user the current logged items

macros.**shutter_manual**()
Set 1-ID fast shutter so that it will not be controlled by the GE TTL signal and can be manually opened and closed with Copen() and Cclose()

macros.**shutter_sweep**()
Set 1-ID fast shutter so that it will be controlled by an external electronic control (usually the GE TTL signal)

macros.**specdate**()
format current date/time as produced in Spec

Returns the current date/time as a string, formatted like "Thu Oct 04 18:24:14 2012"

Example:

```
>>> macros.specdate()
'Thu Oct 11 16:16:39 2012'
```

macros.**write_logging_header**(*filename=''*)
Write a header for parameters recorded when write_logging_parameters() is called.

Parameters **filename** (*str*) – a filename to be be used for output. If not specified, the output is sent to the terminal window.

see *Logging* for an example of use.

macros.**write_logging_parameters**(*filename=''*)
Record the current value of all items tagged to be recorded in add_logging_PV(), add_logging_Global(), add_logging_PVobj(), add_logging_motor() or add_logging_scaler().

Parameters **filename** (*str*) – a filename to be be used for output. If not specified, the output is sent to the terminal window.

see *Logging* for an example of use.

CHAPTER

THREE

MODULE GE: GE IMAGE PROCESSING

This is a module for reading files from the GE angiography detector in use at sector 1

3.1 Overview

Routines	Description
Count_Frames()	Determine the number of frames in a GE image file
getGEimage()	Read a single entire GE image file
getGE_ROI()	Read a section (region of interest) of a GE image file
PlotGEimage()	Plot an image or ROI
sumGE_ROIs()	Report the average intensity for ROIs in a GE image frame
sumAllGE_ROIs()	Reports the average intensity for ROIs for all frames in a file
PlotROIsums()	Plots the ROIs values from sumAllGE_ROIs()

3.2 Complete Function Descriptions

The functions available in this module are listed below.

GE.**Count_Frames**(*filename*)

Determine the number of frames in a GE file by looking at the file size.

Parameters **filename** (*str*) – The filename containing the as-recorded GE images

Returns the number of frames (int).

Example:

```
>>> ifil = '/Users/toby/software/work/1ID/data/AZ91_01306.ge2'
>>> GE.Count_Frames(ifil)
220
```

GE.**PlotGEimage**(*img*, *title*, *tablbl*, *plotlist*, *region=None*, *size=(700, 700)*, *imgwin=None*)

Create a plot of an image in tabbed window

Parameters

- **img** (*array*) – An image, as a numpy array or matplotlib compatible object. Usually this will be created by getGEimage() or getGE_ROI().
- **title** (*str*) – A string with a title for the window
- **tablbl** (*str*) – A string with the title for the new tab (should be short)

- **plotlist** (*list*) – A list of _ImagePlot objects. As new plots are created in this routine they are added to this list. The list is used to assign color maps.
- **region** (*list*) – A list for four numbers which describes the ROI location for use in adding offsets for the plot axes labeling. The numbers are:

element #	label	description
0	xmid	x value for central pixel
1	ymid	y value for central pixel
2	xwid	half-width of ROI in pixels
3	ywid	half-width of ROI in pixels

  The default is to label the pixels starting from zero.
- **size** (*list*) – A list, tuple or wx.size object with the size of the window to be created in pixels. The default is (700,700)
- **imgwin** (*object*) – A plotnotebook object that has been created using `plotnotebook.MakePlotWindow()`, usually in a prior call to `PlotGEimage()`. A value of None (default) causes a new frame (window) to be created.

Returns A reference to the plot window (a plotnotebook object), which will be either imgwin or the new one created in `plotnotebook.MakePlotWindow()`.

Examples:

```
>>> import plotnotebook
>>> import GE
>>> plotlist = []
>>> ifil = '/Users/toby/software/work/1ID/data/AZ91_01306.ge2'
>>> img = GE.getGEimage(ifil,2)
>>> imgwin = GE.PlotGEimage(img,'image window','full image',plotlist)
>>> plotnotebook.ShowPlots()

>>> import plotnotebook
>>> import GE
>>> plotlist = []
>>> ifil = '/Users/toby/software/work/1ID/data/AZ91_01306.ge2'
>>> ROI = GE.getGE_ROI(ifil,2,(100,200,5,7))
>>> imgwin = GE.PlotGEimage(ROI ,'','ROI',plotlist, (100,200,4,6))
>>> plotnotebook.ShowPlots()
```

GE.**PlotROIsums**(*datarray*, *tablbl='ROIs'*, *title=''*, *captions=None*, *size=(700, 700)*, *imgwin=None*)

Plots a series of ROIs

Parameters

- **datarray** (*array*) – a list of MxN array of average intensity values, as returned by `sumAllGE_ROIs()`, where M is the number of frames and N is the number of ROI region(s).
- **tablbl** (*str*) – A string with the title for the new tab. (Should be short; default is "ROIs".)
- **title** (*str*) – A string with a title for the window. Defaults to blank.
- **captions** (*list*) – A list of N strings, where each string specifies a legend caption for each of the N ROI regions. (Default is "ROI #".)
- **size** (*list*) – A list, tuple or wx.size object with the size of the window to be created in pixels. The default is (700,700)

- **imgwin** (*object*) – A plotnotebook object that has been created using `plotnotebook.MakePlotWindow()`, usually in a prior call to `PlotGEimage()`. A value of None (default) causes a new frame (window) to be created.

Returns A reference to the plot window (a plotnotebook object), which will be either imgwin or the new one created in `plotnotebook.MakePlotWindow()`.

Example:

```
>>> regionlist = [(1335,1525,50,50),(1435,1525,50,50),
...               (335,1525,50,50),(1935,1525,50,50)]
>>> caps = [str(i[0])+','+str(i[1]) for i in regionlist]
>>> ROIarr = GE.sumAllGE_ROIs(imgfile,regionlist)
>>> GE.PlotROIsums(ROIarr, captions=caps)
>>> import plotnotebook
>>> plotnotebook.ShowPlots()
```

GE.**getGE_ROI** (*filename*, *frame*, *region*)

Read a section (region of interest) of a GE image from a file. This is usually faster than reading an entire image.

Parameters

- **filename** (*str*) – The filename containing as-recorded GE images
- **frame** (*int*) – the image number on the file, counted starting at 1
- **region** (*list*) – describes the region to be extracted

element #	label	description
0	xmid	x value for central pixel
1	ymid	y value for central pixel
2	xwid	half-width of ROI in pixels
3	ywid	half-width of ROI in pixels

The extracted ROI will be pixels img[ymid-ywid:ymid+ywid,xmid-xwid:xmid+xwid] where img is the full image.

Returns An image as a (2*ywid)x(2*xwid) numpy memmap (behaves like an array) of intensities

Example:

```
>>> ifil = '/Users/toby/software/work/1ID/data/AZ91_01306.ge2'
>>> GE.getGE_ROI(ifil,2,(100,200,4,6))
memmap([[1755, 1762, 1763, 1761, 1766, 1762, 1761, 1756],
   [1761, 1763, 1760, 1764, 1765, 1769, 1755, 1758],
   [1762, 1762, 1763, 1758, 1769, 1769, 1757, 1756],
   [1760, 1767, 1764, 1763, 1763, 1765, 1762, 1756],
   [1760, 1764, 1760, 1763, 1763, 1762, 1758, 1758],
   [1761, 1760, 1766, 1762, 1761, 1767, 1761, 1761],
   [1754, 1761, 1765, 1754, 1760, 1768, 1760, 1759],
   [1763, 1764, 1764, 1763, 1766, 1762, 1765, 1761],
   [1760, 1757, 1761, 1765, 1766, 1766, 1761, 1759],
   [1761, 1761, 1761, 1761, 1761, 1763, 1757, 1758],
   [1757, 1765, 1760, 1767, 1764, 1768, 1758, 1760],
   [1762, 1765, 1764, 1760, 1764, 1766, 1761, 1761]], dtype=uint16)
```

GE.**getGEimage** (*filename*, *frame*)

Read a single entire GE image from a file

Parameters

- **filename** (*str*) – The filename containing as-recorded GE images

- **frame** (*int*) – the image number on the file, counted starting at 1 An exception is raised if frame is greater than the number of frames in the file.

Returns An image as a 2048x2048 numpy array of intensities

Example:

```
>>> ifil = '/Users/toby/software/work/1ID/data/AZ91_01306.ge2'
>>> GE.getGEimage(ifil,2)
array([[1699, 1713, 1713, ..., 1701, 1697, 1695],
   [1708, 1717, 1717, ..., 1708, 1703, 1705],
   [1715, 1719, 1719, ..., 1708, 1707, 1707],
   ...,
   [1714, 1720, 1714, ..., 1698, 1702, 1697],
   [1714, 1718, 1716, ..., 1702, 1703, 1702],
   [1701, 1704, 1697, ..., 1684, 1685, 1687]], dtype=uint16)
```

GE.**sumAllGE_ROIs** (*filename*, *regionlist*, *processes=1*)

Computes the average intensity for each ROI specified in the regionlist for every frame in a raw GE image file.

param str filename The filename containing as-recorded GE images

param list regionlist A list of ROI regions. Each ROI region is consists of 4 elements:

element #	label	description
0	xmid	x value for central pixel
1	ymid	y value for central pixel
2	xwid	half-width of ROI in pixels
3	ywid	half-width of ROI in pixels

param int processes specifies the number of simultaneous processes that can be used to perform ROI integration using the Python multiprocessing module. The default, 1, will not use this module and all computations are done in the current thread.

returns a list of MxN array of average intensity values, where M is the number of frames and N is the number of ROI region(s) in regionlist.

Examples:

```
>>> ifil = '/Users/toby/software/work/1ID/data/AZ91_01306.ge2'
>>> GE.sumAllGE_ROIs(ifil, [(100,200,4,6), (1335,1525,50,50)])
array([[ 1794.57291667,  1801.2036    ],
   [ 1761.80208333,  1792.6894    ],
   [ 1760.5       ,  1791.7353    ],
   [ 1760.36458333,  1791.4961    ],
   [ 1760.03125   ,  1791.6162    ],
   ...
   [ 1760.0625    ,  1779.0867    ],
   [ 1759.72916667,  1779.1182    ],
   [ 1759.5       ,  1779.2508    ]])
```

In the example above, two ROIs are integrated for all frames in a file in the current Python interpreter.

```
import GE
import numpy as np
import time
imgfile = '/tmp/AZ91_01306'
regionlist = [(1335,1525,50,50),(1435,1525,50,50),
              (335,1525,50,50),(1935,1525,50,50)]
nframe = GE.Count_Frames(imgfile)
l = {}
```

```
for proc in range(10):
    st = time.time()
    l[proc] = GE.sumAllGE_ROIs(imgfile,regionlist, proc)
    print 'sec per frame, processors=',proc,(time.time()-st)/float(nframe)
    assert(np.allclose(l[0],l[proc]))
```

The example above integrates 4 ROIs and compares running with all comutations in the current Python thread (processes=0 and 1) with running with up to 9 concurrent processes. Usually one sees a speed-up with ~1.5 times the actual number of cores for multiprocessing. The assert is used to confirm the computation returns the same results independent of the number of processes.

GE.**sumGE_ROIs**(*filename*, *frame*, *regionlist*)

Reads a frame from a raw GE image file and returns a list of the average intensity for each ROI specified in the regionlist.

Parameters

- **filename** (*str*) – The filename containing as-recorded GE images
- **frame** (*int*) – the image number on the file, counted starting at 1
- **regionlist** (*list*) – A list of ROI regions. Each ROI region is consists of 4 elements:

element #	label	description
0	xmid	x value for central pixel
1	ymid	y value for central pixel
2	xwid	half-width of ROI in pixels
3	ywid	half-width of ROI in pixels

Returns a list of N average intensity values, one for each ROI region in regionlist.

Example:

```
>>> ifil = '/Users/toby/software/work/1ID/data/AZ91_01306.ge2'
>>> regionlist = [(1335,1525,50,50),(1435,1525,50,50),]
>>> GE.sumGE_ROIs(ifil,2,regionlist)
[1792.6894, 1780.4342]
```

GE.**sumGE_ROIs_wrapper**(*args*)

Provides an interface to `sumGE_ROIs()` that allows it to be called with a single argument.

INDEX

R

S

U

W